Tema 52: Koristite subprocess da biste upravljali procesima potomcima

Python ima prekaljene biblioteke za izvršavanje procesa potomaka i upravljanje njima. Zbog toga je to odličan jezik za međusobno spajanje drugih alata, kao što su pomoćni programi komandne linije. Kada postojeći komandni skriptovi postanu komplikovani, što se često dešava tokom vremena, prirodno je preraditi ih zarad čitkosti i održivosti.

Procesi potomci (engl. *child processes*) započeti u Pythonu mogu se izvršavati paralelno, omogućavajući vam da iskoristite Python da biste utrošili sva CPU jezgra mašine i maksimirali propusni opseg programa. Iako sâm Python može biti ograničen CPU-om (Tema 53: Koristite niti za blokiranje U/I, a izbegavajte za paralelnost), lako je koristiti Python za pokretanje poslova koji intenzivno koriste CPU i za upravljanje njima.

Python ima mnogo načina da pokreće potprocese (npr, `os.popen`, `os.exec*`), ali najbolji izbor za upravljanje procesima potomcima jeste korišćenje ugrađenog modula `subprocess`. Pokretanje procesa potomka pomoću `subprocess` je jednostavno. Ovde koristim pomoćnu funkciju `run` iz tog modula da bih pokrenuo proces, pročitao njegov rezultat i proverio da li se on čisto završio:

```
import subprocess

result = subprocess.run(
    ['echo', 'Hello from the child!'],
    capture_output=True,
    encoding='utf-8')

result.check_returncode()  # Nema izuzetka znači čist izlaz
print(result.stdout)

>>>
Hello from the child!
```

> **Napomena**
>
> Primeri u ovoj temi pretpostavljaju da su u vašem sistemu dostupne komande `echo`, `sleep` i `openssl`. U Windowsu to možda nije slučaj. Pogledajte kompletan kod primera iz ove Teme da biste videli konkretna uputstva kako se ovi delovi koda izvršavaju u Windowsu.

Procesi potomci izvršavaju se nezavisno od roditeljskog procesa, Pythonovog intepretera. Ako napravim potproces koristeći klasu `Popen` umesto funkcije `run`, mogu periodično da anketiram status procesa potomka dok Python obavlja ostali posao:

```
proc = subprocess.Popen(['sleep', '1'])
while proc.poll() is None:
    print('Working...')
    # Ovde je posao koji zahteva dosta vremena
    ...

print('Exit status', proc.poll())

>>>
Working...
Working...
Working...
Working...
Exit status 0
```

Razdvajanje procesa potomka od roditelja oslobađa roditeljski proces za paralelnost više procesa potomaka. Ovde to radim tako što započinjem sve procese potomke zajedno, zadajući `Popen` unapred:

```
import time

start = time.time()
sleep_procs = []
for _ in range(10):
    proc = subprocess.Popen(['sleep', '1'])
    sleep_procs.append(proc)
```

Kasnije, želim da oni završe svoje ulaze/izlaze i prekinu se pomoću metode `communicate`:

```
for proc in sleep_procs:
    proc.communicate()

end = time.time()
delta = end - start
print(f'Finished in {delta:.3} seconds')

>>>
Finished in 1.05 seconds
```

Da su se ovi procesi izvršavali jedan za drugim, ukupno odlaganje bilo bi 10 ili više sekundi, umesto oko jedne sekunde koliko sam izmerio.

Podatke možete i usmeravati iz Pythonovog programa u potproces i pozivati njegov rezultat. Zahvaljujući tome, više drugih programa može da radi paralelno. Na primer, recimo da želim da koristim alat komandne linije `openssl` da bih šifrovao neke podatke. Vrlo lako ću započeti proces potomak sa argumentima komandne linije i kanalima U/I:

```
import os
def run_encrypt(data):
    env = os.environ.copy()
    env['password'] = 'zf7ShyBhZOraQDdE/FiZpm/m/8f9X+M1'
    proc = subprocess.Popen(
        ['openssl', 'enc', '-des3', '-pass', 'env:password'],
        env=env,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE)
    proc.stdin.write(data)
    proc.stdin.flush()  # Pobrinite se da potomak dobije ulaz
    return proc
```

Ovde usmeravam nasumične bajtove u funkciju za šifrovanje, ali u praksi bi ovaj ulazni kanal dobijao podakte od korisničkog unosa, programa za obradu datoteka, mrežnog ulaza itd:

```
procs = []
for _ in range(3):
    data = os.urandom(10)
    proc = run_encrypt(data)
    procs.append(proc)
```

Procesi potomci se izvršavaju paralelno i koriste svoj ulaz. Ovde čekam da završe, a potom pozivam njihove konačne rezultate. Rezultat su nasumični šifrovani bajtovi, kao što je i očekivano:

```
for proc in procs:
    out, _ = proc.communicate()
    print(out[-10:])
```

```
>>>
b'\x8c(\xed\xc7m1\xf0F4\xe6'
b'\x0eD\x97\xe9>\x10h{\xbd\xf0'
b'g\x93)\x14U\xa9\xdc\xdd\x04\xd2'
```

Moguće je napraviti i lance paralelnih procesa, kao što su UNIX-ovi cevovodi, koji povezuju rezultat jednog procesa potomka sa ulazom za drugi i tako redom. Evo funkcije koja pokreće alat komandne linije `openssl` kao potproces da bi generisala heš funkciju Whirlpool ulaznog toka:

```
def run_hash(input_stdin):
    return subprocess.Popen(
        ['openssl', 'dgst', '-whirlpool', '-binary'],
        stdin=input_stdin,
        stdout=subprocess.PIPE)
```

Sada mogu da pokrenem jedan skup procesa da šifriraju podatke, a drugi skup procesa da naknadno heširaju njihov šifrovani rezultat. Videćete da sam morao da budem oprezan sa načinom na koji se instanca `stdout` „uzvodnog“ procesa zadržava u procesu Pythonovog interpretera koji započinje ovaj cevovod procesa potomaka:

```
encrypt_procs = []
hash_procs = []
for _ in range(3):
    data = os.urandom(100)

    encrypt_proc = run_encrypt(data)
    encrypt_procs.append(encrypt_proc)

    hash_proc = run_hash(encrypt_proc.stdout)
    hash_procs.append(hash_proc)

    # Brine se da potomak utroši ulazni tok i
    # da metoda communicate() ne ukrade slučajno
    # ulaz od potomka. Dozvoljava i da se SIGPIPE proširi na
    # uzvodni proces ako se nizvodni prekine.
    encrypt_proc.stdout.close()
    encrypt_proc.stdout = None
```

U/I između procesa potomaka dešava se automatski kada se oni pokrenu. Sve što treba da radim jeste da sačekam da se oni završe i ispišu konačan rezultat:

```
for proc in encrypt_procs:
    proc.communicate()
    assert proc.returncode == 0

for proc in hash_procs:
    out, _ = proc.communicate()
    print(out[-10:])
    assert proc.returncode == 0
```

```
>>>
b'\xe2j\x98h\xfd\xec\xe7T\xd84'
b'\xf3.i\x01\xd74|\xf2\x94E'
b'5_n\xc3-\xe6j\xeb[i'
```

Ukoliko sam zabrinut da se procesi potomci nikada neće završiti ili da će nekako blokirati ulazne ili izlazne kanale, mogu da prosledim parametar `timeout` metodi `communicate`. Zahvaljujući tome, biće generisan izuzetak ukoliko proces potomak nije završio u određenom vremenskom periodu, pružajući mi priliku da prekinem potproces koji se loše ponaša:

```
proc = subprocess.Popen(['sleep', '10'])
try:
    proc.communicate(timeout=0.1)
except subprocess.TimeoutExpired:
    proc.terminate()
    proc.wait()

print('Exit status', proc.poll())

>>>
Exit status -15
```

Zapamtite

- Koristite modul `subprocess` da biste izvršavali procese potomke i upravljali njihovim ulaznim i izlaznim tokovima.
- Procesi potomci se izvršavaju paralelno sa Pythonovim interpreterom, omogućavajući vam da maksimirate iskorišćenost CPU jezgara.
- Koristite pomoćnu funkciju `run` za jednostavne namene, a klasu `Popen` za napredne namene kao što su cevovodi u stilu UNIX-a.
- Koristite parametar `timeout` metode `communicate` da biste izbegli blokade i zaostale procese potomke.

Tema 53: Koristite niti za blokiranje U/I, a izbegavajte za paralelnost

Standardna implementacija Pythona zove se CPython. CPython izvršava Pythonov program u dva koraka. Prvo, on raščlanjuje i prevodi izvorni tekst u *binarni kod* (engl. *bytecode*), predstavu programa niskog nivoa u vidu 8-bitnih uputstava. (Od Pythona 3.6, to je zapravo *wordcode* sa 16-bitnim uputstvima, ali zamisao je ista.) Potom, CPython izvršava binarni kod koristeći interpreter zasnovan na steku. Interpreter binarnog koda ima stanje koje se mora očuvati i mora biti koherentno dok se Pythonov program izvršava. CPython obezbeđuje koherentnost pomoću mehanizma nazvanog globalno zaključavanje interpretera (engl. *global interpreter lock, GIL*).

U osnovi, GIL je uzajamno isključivo zaključavanje (mutex) koje sprečava da na CPython utiče predupredni višenitni rad, kada jedna nit preuzima kontrolu nad programom prekidajući durgu nit. Takvo prekidanje bi moglo da naruši stanje interpretera (npr, brojač referenci za sakupljanje otpada) ukoliko se desi u neočekivanom trenutku. GIL sprečava takve prekide i stara se da svaka instrukcija binarnog koda radi ispravno sa implementacijom Cpython-a i njegovim modulima za C proširenja.

GIL ima jedno bitno negativno sporedno dejstvo. Sa programima napisanim na jezicima kao što su C++ i Java, višestruke niti izvršavanja znače da bi program mogao da koristi više CPU jezgara istovremeno. Mada Python podržava više niti izvršavanja, GIL dovodi do toga da samo